Рассмотрим следующий важный момент при работе с классами. Компилятор по умолчанию использует в классе свой набор следующих методов:

- конструктор по умолчанию;
- конструктор копирования;
- операция присваивания копированием;
- деструктор (в виде заглушки).

А, начиная со стандарта C++11, к ним добавилось еще два:

- конструктор перемещения;
- операция присваивания перемещением.

Конструктор перемещения и операцию присваивания перемещением мы с вами их оставим в стороне. Нас пока будет интересовать самый базовый набор методов по умолчанию.

Первый вопрос, почему они вообще существуют? Очевидно, что без них не возможно создание и работа с объектами классов или структур. Например, объявляя простой класс:

```language-cpp
class Vector {
    int x {0};
    int y {0};
};
```

Мы сразу получаем базовый функционал:

```language-cpp
int main()
{
    Vector v1;      // конструктор по умолчанию
    Vector v2(v1);  // конструктор копирования
    Vector v3;
    v3 = v1;        // операция присваивания копированием

    return 0;
}
```

И все благодаря наличию у компилятора набора методов класса, используемых по умолчанию. Но мы уже знаем, что все эти методы могут быть замещены собственными, явно прописанными в классе. Причем, если в классе явно объявить любой конструктор (отличный от конструктора

копирования), то конструктор по умолчанию перестает существовать. Например:

```language-cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector(int a, int b): x(a), y(b)
        { }
};
```

Обратите внимание, что конструктор с двумя параметрами – это не конструктор по умолчанию. Тем не менее, компилятор более не станет использовать конструктор без параметров по умолчанию. Он, как бы, перестает существовать для класса Vector. А вот конструктор копирования и операция присваивания копирования при этом никуда не пропали.

Если же вместо конструктора с двумя параметрами объявить конструктор копирования:

```language-cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector(const Vector& other)
    {
        x = other.x;
        y = other.y;
    }
};
```

то будет замещен и конструктор копирования и конструктор по умолчанию (без параметров). Останется только стандартная операция присваивания копированием. В результате, объект класса Vector создать командой:

```language-cpp
Vector v1;
```

станет невозможно. Однако поправить это достаточно просто. В классе Vector указать компилятору продолжать использовать свой собственный конструктор по умолчанию:

```language-cpp
Vector() = default;
```

Здесь ключевое слово default, как раз и предписывает компилятору подставлять (при необходимости) вызов встроенного конструктора без параметров. Либо, как вариант, мы можем объявить свой собственный конструктор по умолчанию:

```cpp
Vector() {};
```

В данном случае различий в работе класса никаких не будет.

**Забегая вперед, отмечу, что операцию присваивания мы также можем переопределить (заменить) своей собственной**. Сделать это можно следующим образом:

```cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector() {};
    Vector(const Vector& other)
    {
        x = other.x;
        y = other.y;
    }

    const Vector& operator=(const Vector& other)
    {
        if(this == &other) return *this;

        this->x = other.x;
        this->y = other.y;
        return *this;
    }
};
```

Переопределение операции присваивания никак не влияет на сокрытие встроенных (стандартных) конструкторов класса.

## Ключевое слово delete

При желании мы можем явно отказаться от использования в классе того или иного стандартного конструктора. Для этого, начиная со стандарта C++11, вводится ключевое слово delete, с помощью которого можно помечать определенный конструктор следующим образом:

```cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector() = delete;
};
```

В этом случае, в классе Vector остается конструктор копирования, но отменяется конструктор по умолчанию. Соответственно, объекты этого класса не могут быть созданы стандартным образом:

```cpp
Vector v1;  // ошибка, нет конструктора по умолчанию
```

Но, если пометить конструктор копирования, как удаленный:

```cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector(const Vector& other) = delete;
};
```

то компилятор откажется от использования и конструктора копирования и конструктора по умолчанию. Выйти из этой ситуации можно следующим образом:

```cpp
class Vector {
    int x {0};
    int y {0};

public:
    Vector() = default;
    Vector(const Vector& other) = delete;
};
```

Тогда будет использоваться стандартный конструктор по умолчанию, но отсутствовать конструктор копирования.

## Сокрытие конструкторов

Еще одним распространенным на практике способом управления созданием объектов является определение конструкторов в приватной секции.

Например, если определить приватный конструктор копирования и публичный конструктор по умолчанию:

```language-cpp
class Vector {
    int x {0};
    int y {0};

private:
    Vector(const Vector& other) = default;
public:
    Vector() = default;
};
```

то, очевидно, объекты класса можно совершенно спокойно создавать, но нельзя будет инициализировать другими подобными объектами.

Если оба конструктора сделать приватными:

```language-cpp
class Vector {
    int x {0};
    int y {0};

private:
    Vector() = default;
    Vector(const Vector& other) = default;
};
```

то объекты этого класса можно будет создавать только внутри методов этого класса. **Таким образом, используя секцию private, мы можем управлять возможностью создания и копирования объектов определенного класса**. Эта возможность не редко применяется в практике программирования.